ChatGPT

Vibe-Steering Application Development Workflow

Overview: What is *Vibe Steering*?

“Vibe steering” refers to leveraging AI assistance and best practices to **steer** the development of an application in a desired direction or *vibe*. Unlike ad-hoc “vibe coding” (simply prompting an AI to build something and hoping for the best), vibe steering is a **systematic workflow**. It combines your full-stack engineering skills with AI tools (like Claude) to save time on tedious coding while ensuring the app is well-structured, maintainable, and visually appealing. This workflow addresses common pitfalls of naive AI-generated code – such as inconsistent results and unmaintainable code – by keeping a human in the loop and following best practices [1] [2]. The goal is to enjoy the productivity benefits of AI (“building features from descriptions” [3], automating boilerplate tasks [4], etc.) **without** sacrificing code quality or design polish.

Key Principles of Vibe Steering:

- **Human-Guided AI** – Use AI to generate code or designs from natural language prompts, but always guide its output with your expertise (review, test, and refine the AI’s work). You remain the “driver” steering the vibe.
- **Best Practices & Tech Standards** – For each project, select tech stacks and architecture patterns that are considered best-in-class for that domain. This provides a solid foundation that the AI can build upon correctly.
- **Consistency and Reproducibility** – Establish templates for documentation and code style so that even if AI contributions are non-deterministic, the project stays consistent. Version control every AI-generated change to avoid “mysterious” code that no human understands [2].
- **Design Frameworks for Visuals** – Incorporate UI component libraries or design systems to ensure a good look and feel, since design isn’t your strong suit. As one developer put it, *“Good design is harder than it looks… if you don't have a design system … just use a component library”* [5]. This way, the app’s vibe (UI/UX) is appealing out-of-the-box, and AI can help fine-tune on top of a solid base.

Project Definition and Vibe Discovery

Start every project by clearly defining what you’re building and the desired “vibe” or user experience. This phase is all about understanding the **purpose, audience, and style** of the application before writing any code. By pinning down the vision early, you can instruct AI tools more effectively and make coherent design decisions.

Ask Yourself (Project Kick-off Questions):

- *What problem does this application solve, and who are the end users?* (e.g. a casual consumer app vs. an enterprise tool – the vibe will differ)
- *What mood or visual style should the app convey?* (e.g. playful and colorful, or sleek and professional? Think of adjectives or references.)

- *What are the core features or user stories?* (High-level summary of what the app will do)
- *Are there any specific design inspirations or existing apps with a vibe you want to emulate?* (Gather a few examples or templates if possible)

Take notes on these answers in a **Project Brief** document (this can be a simple Markdown file in the project directory, like `PROJECT_BRIEF.md`). This brief will guide both you and the AI in the next steps. For instance, noting a desired vibe like "minimalist, dark theme, developer-oriented" will help when prompting AI to generate UI components or CSS later.

Templatized Documentation: Start drafting a README or project overview document *before* coding. Embrace "README-driven development," where you outline the project's goals, features, and usage in the README first [6]. This forces you to think through the app's flow and ensures the project has a solid documentation foundation from day one. *Write it before you write a single line of code. This way you can plan your project better and be confident that your project already has a great foundation in its documentation* [6]. In this initial README, include sections like **Purpose**, **Features**, **Tech Stack (to be decided)**, and **Design Goals** – you will fill in details as you go.

Tech Stack Selection (Best Practices per Project)

Each project might use a different tech stack, but the guiding rule is to choose **proven, best-practice technologies** that suit the application type. Since you're comfortable learning new stacks, focus on what's industry-standard and well-supported (this also makes it easier for AI assistants to generate conventional, correct code).

Determine the Stack – Key Considerations:

- **Application Type:** Is this a web application, mobile app, desktop tool, CLI, etc.? *Example:* For a web **SPA** you might choose React or Next.js; for a mobile app, maybe Flutter or React Native; for a data-heavy backend, maybe Python with FastAPI or Node with Express.
- **Best-in-Class Tools:** Research what the current best practices are for that type of project. *For instance:* A modern web app might use a React/Vue front-end with a component library, a Node/Express or Django/FastAPI backend, a PostgreSQL database, and containerization for deployment. If unsure, do a quick survey of recent projects or frameworks popular in that domain.
- **Your Expertise and Team:** If it's just you, lean towards stacks you know or that have lots of documentation/community support (AI will have more training info on popular frameworks too). If working with others, consider common team knowledge. Either way, avoid extremely niche frameworks for which neither you nor the AI will find much help.
- **Integration and DevOps:** Since you have DevOps/Kubernetes skills, also consider how the stack will run and deploy. For example, ensure the chosen tech can be easily containerized. Many modern stacks have official Docker images or deployment guides, which simplifies later steps.

Once decided, update the README's **Tech Stack** section with your choices and rationale. Keep it flexible – you might adjust as requirements evolve. It's also helpful to note version numbers or particular configurations (e.g. "Node.js 20 + Express 5, using TypeScript") to keep consistency.

Leverage AI for Research: You can use Claude or another assistant to compare stacks or get quick summaries of pros/cons. For example: *"Claude, what are the pros and cons of using Next.js for a SaaS* *dashboard app?"* This can surface considerations you hadn't thought of. Just be sure to verify any claims (the assistant might hallucinate details), and make the final choice yourself.

Project Setup and Template Structure

With the project's vision and stack defined, create a standardized project structure. The idea is to have a **template directory** that you can reuse for any new project, containing common files and scaffold. This template will include both **templatized files** (where you fill in project-specific details) and **generic boilerplate** that applies to most projects.

Base Directory Layout:

```
project-name/
├── README.md                # Project overview (start filled by template, then
edited)
├── docs/                    # Documentation (e.g. design docs, user manual,
etc.)
│   └── DESIGN.md            # (Optional) design/UX guidelines for the project
├── src/                     # Main source code directory (structure depends on
stack)
├── .claude/                 # Claude-related configuration and skills
│   ├── skills/              # Custom Claude skills for this project
│   └── settings.yml         # (If needed, Claude config file)
├── .gitignore
├── docker/                  # Dockerfiles or deployment scripts (if applicable)
└── tests/                   # Test suites
```

Feel free to modify according to project needs (for example, add a `backend/` and `frontend/` folder if it's a full-stack web app, etc.). The key is to **start with a skeleton** so you're not starting from scratch. You can maintain a "starter template" repo on your machine or GitHub, and copy it when beginning a new app, then rename things accordingly.

Templated Files to Include:

- `README.md` – A template README with placeholders for Project Name, Description, Stack, How to Run, etc. Since you likely started drafting this in the previous phase (Project Definition), your template can provide a standard outline (Introduction, Features, Setup Instructions, Usage, Contributing, License). You will edit this as the project evolves.
- `docs/DESIGN.md` – If visual design or complex architecture is involved, use this to document UI decisions and architecture diagrams. For example, if you choose a color scheme or UX principle (the "vibe" definition), write it here for reference. This helps maintain consistency later.
- **Configuration Files** – Standard configs like `.gitignore` for your stack, a basic `package.json` (for Node projects) or `pyproject.toml` (for Python) with project name, a `Dockerfile` or `docker-compose.yml` if relevant (perhaps as simple templates you'll fill out when ready to containerize).

- **CI/CD Pipeline Config** – (Optional) If you have a preferred CI setup (GitHub Actions, GitLab CI, etc.), include a template YAML that runs tests and linting. This can be set up later, but having a placeholder reminds you to implement CI early.
- `.claude/skills/` – Include this directory even if empty initially. This is where you'll add Claude Skills to automate tasks or extend Claude's abilities for the project. For example, you might add a skill for quickly spinning up boilerplate or running the app.

Using Claude Tools for Initial Scaffolding: Claude's coding assistant can actually help create the initial project files. For instance, you can open VS Code with Claude extension, create an empty `package.json`, and ask Claude to fill it with a template for a Node project (or use `npm init` via Claude's terminal access). Claude Code is designed to "take action" – it can directly create and edit files and even run commands [7]. So you could prompt: *"Create a basic Express app structure with TypeScript, with a Dockerfile and a GitHub Actions workflow for CI"*. Claude might draft those files for you. Always review what it writes, but this can jump-start the setup significantly.

Development Workflow with AI Assistance

Now comes the core of building the application. **Your goal is to implement features systematically while using AI to reduce tedious work.** You'll iterate through cycles of writing code (with AI help), testing, and refinement. Throughout, keep the "vibe" in mind – both in terms of code quality and user experience.

General Iterative Approach:

1. **Plan the Next Feature/Task:** Before coding, describe in natural language what you're about to build (e.g., "user login form with OAuth", "image gallery component", "CI pipeline for running tests on push"). This can be a short paragraph or bullet list of requirements. If it helps, write this in a `tasks.md` or as comments in code.
2. **Use Claude to Generate a Plan & Code:** Leverage Claude's capability to *"make a plan, write the code, and ensure it works"* from a description [3]. For example, in VSCode you could prompt Claude: *"Create a new React component for a login form with email/password fields and basic validation. Use Material-UI for styling."* Claude will typically respond with a plan (outlining steps or file changes) and propose code. It might even open new files in your editor with the generated content.
3. **Tip:** For complex features, ask Claude to outline a plan first (e.g., "Plan the steps to implement feature X") then confirm before writing code. Claude Code has a "Plan Mode" for safe analysis that you can use for big refactors or critical code generation [8].
4. **Review and Edit the AI-Generated Code:** Treat AI output as a junior developer's code – always review it. Ensure it matches your intended functionality and style. Because you chose established frameworks and best practices, the AI's suggestions are more likely to be aligned with known patterns. Still, verify things like security (e.g., did it handle input sanitization?) and performance (e.g., not doing something horribly inefficient).
5. **Test Immediately:** Run the code or at least its unit tests (if available) as soon as it's generated. Claude can run commands in the terminal [9], so you could have it execute the app or run `npm test` for instance. If an error occurs, feed the error back to Claude – *"Here's the stack trace, what went wrong?"*. Claude can help debug and fix issues, analyzing your codebase to identify the problem [10] [11].

6. **Iterate:** After a feature works, commit the changes (include meaningful commit messages). Update documentation if needed (the README or docs). Then move on to the next feature or improvement. This iterative loop continues until the application is complete.

Automating Tedious Tasks: One huge benefit of vibe steering is offloading grunt work to AI. Claude Code can **"automate tedious tasks"** like fixing lint errors, converting formatting, or updating boilerplate across files [4]. For example: - If you need to rename a variable or refactor code in multiple files, you can instruct Claude to do it project-wide. - To generate repetitive code (say, similar CRUD endpoints or similar UI components), consider writing a Claude skill or using a loop prompt. *E.g.:* "Claude, for each model in `models.json`, create a CRUD controller." It can script this reliably if given a pattern. - Use **custom Claude Skills** for frequent tasks. Skills are essentially stored AI instructions you can invoke with a slash command [12]. For instance, you could create a skill like `/scaffold-component` that takes a component name and description and generates a boilerplate React component with that name. The skill's `SKILL.md` might define triggers so Claude can even auto-suggest using it when you talk about creating new components. (Skills consist of a YAML frontmatter with a name/description and the instructions to execute [13] – you'll create them in `.claude/skills/` and Claude will pick them up). Over time, your personal library of skills can grow, further speeding up routine coding tasks.

Maintaining Control and Quality: Always remember that *you* are the senior engineer and Claude is an assistant. If its output doesn't meet your standards, you can ask for a revision or simply edit manually. Don't hesitate to scrap an AI-generated solution if it feels off and try a different approach or prompt. The workflow is meant to save time, but not at the expense of doing things correctly. By steering with precise prompts and using your judgment, you prevent the codebase from turning into the kind of "mutating monster" that critics of vibe coding warn about [2].

Designing the User Experience (Visual "Vibe")

Since one goal is a visually appealing app (despite design not being your forte), take advantage of frameworks and AI to **achieve a good UI/UX with minimal manual design work**.

Use Component Libraries / UI Frameworks: As noted earlier, using a well-designed component library can instantly improve your app's look and feel. Libraries like **Material UI, Ant Design, Bootstrap, Tailwind UI,** or design systems like **IBM Carbon** can provide polished, accessible components. This saves you from having to design from scratch. In fact, experienced developers advise that unless you have a solid design system or deep CSS skills, *"just use a component library"* to avoid inconsistent, poor design [5].

- For web projects: consider installing a library such as Material-UI or Chakra UI for React, or using Tailwind CSS with pre-designed components (there are many open-source Tailwind component collections).
- For mobile projects: use the native platform's design guidelines (Material on Android, Cupertino on iOS) via frameworks like Flutter or React Native libraries, so that the default look is already good.
- For any UI: maintain a **style guide** (could be a section in `docs/DESIGN.md`) where you note your primary color palette, font choices, spacing guidelines, etc. You can pick these from the library's theme or have AI help generate them (e.g., "Claude, suggest a color scheme with a calm vibe, including primary, secondary, and accent colors").

AI-assisted Design Tweaks: With Claude in VSCode, you can iterate on the UI by describing the changes you want: - *"Make the header section have a hero image background with overlay text in white."* – Claude can write the CSS or code for that. - *"The form looks plain; suggest some improvements with Tailwind classes for a more modern look."* – Claude could apply styling changes. - If you have an existing design or template, you can paste a snippet and ask Claude to adjust it (Claude Code can also fetch assets from web if needed, using web tool access [14] [15] for URLs or perhaps images – though for design, it's mostly about code).

Always preview the changes in a browser or emulator. Visual tweaking often requires a few cycles of adjusting margins, colors, etc. Use AI to do the heavy lifting (it can be faster than manually typing CSS), but rely on your eyes for the final say – if something looks off, correct it or prompt again with a different approach.

Responsiveness & Accessibility: Don't forget to ensure the "vibe" reaches all users: - Ask Claude to help make the design responsive: *"Ensure this page is mobile-friendly, using flex or grid where appropriate."* It can adjust your layout code. - Accessibility is crucial for a good UX. If using a component library, much of this is baked in, but still ask: *"Does this form have proper ARIA labels and keyboard navigation?"* Claude can analyze the JSX/HTML and fix accessibility issues as a task (AI can be surprisingly adept at spotting missing alt text or ARIA roles if prompted). - Keep user feedback in mind. If you have beta users or testers, incorporate their feedback about the look and feel in future AI prompts (e.g., "Users said the site is too dark – lighten the theme colors by 20%").

Testing and Quality Assurance

A well-written app isn't just feature-complete; it's also reliable and maintainable. Integrate testing and QA into the workflow from early on, using both traditional tools and AI assistance.

Write Tests (and use AI to help): For each feature or module, write unit tests or integration tests. This can be tedious, but AI is great at boilerplate – you can say, *"Generate a Jest test suite for the login component covering valid and invalid inputs."* Claude can produce a starting point for tests. Ensure you understand the tests and that they truly check the right things. Running the tests will also double-check that the feature works as expected. If a test fails, debug (you can ask Claude *"why is this test failing?"* and it might identify the bug in your code).

Continuous Integration: If you set up a CI pipeline (like GitHub Actions or similar), use it to run tests and linters on each commit or PR. Claude can assist in writing the YAML for these workflows if you prompt it. For example, *"Create a GitHub Actions workflow that lints the code with ESLint and runs tests on Node 20"*. It should produce a usable config.

Code Quality – Linters & Formatters: Use linters (ESLint, Pylint, etc. depending on stack) and auto-formatters (Prettier, Black, etc.). These can often be integrated into your editor or run via Claude: - If the linter reports 100 minor issues (unused vars, styling issues), you can ask Claude to fix them in bulk. In fact, Claude Code can fix "fiddly lint issues" in one go [4]. For instance, `/format-code` could be a skill or command to auto-apply prettier or to ask Claude to reformat a file to PEP8 standard, etc. - Enforce consistent style early to avoid large churn later. Include linter configs in your template.

Manual Code Reviews: Even if you're solo, perform pseudo code-reviews. Periodically, read through the AI-generated code as if you were reviewing a contributor's pull request. You might catch subtler issues (inefficient algorithms, unclear naming, etc.). You can also have Claude explain parts of the code to you to ensure *you* fully grok it. For example, *"Explain how the authentication flow works in this code"* – if the explanation (from Claude) or your understanding reveals any gaps or unintended behavior, that's an area to refactor.

Keep an Eye on AI Limitations: Remember that AI might occasionally produce code that just *looks* confident but is slightly off or uses deprecated practices. If something seems odd, double-check official docs or do a quick web search. This is where your background and instincts are important. The workflow encourages using AI, but **not blindly** – always validate.

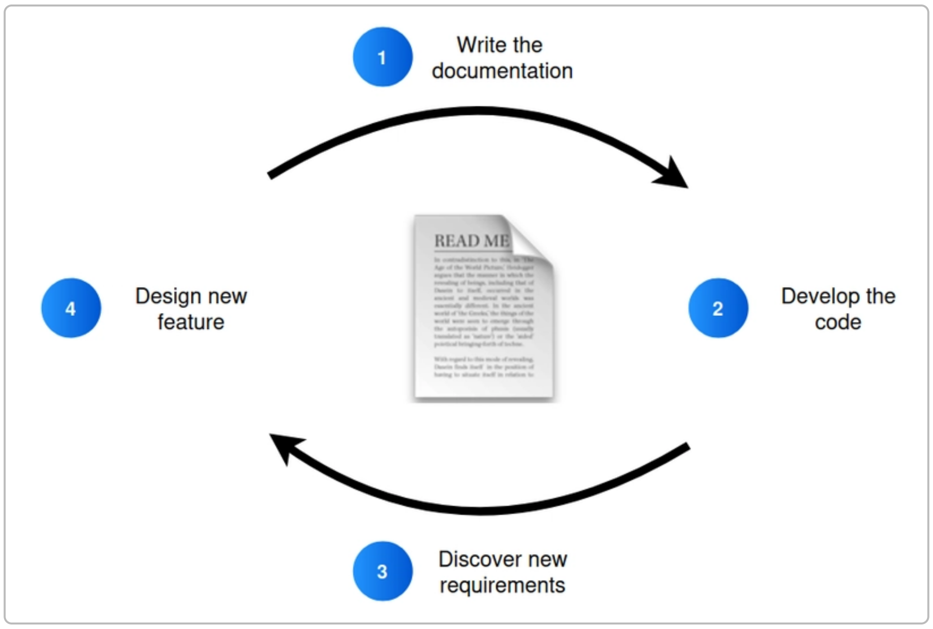
Documentation and Knowledge Management

By the time you've implemented several features, make sure your documentation keeps pace. A systematic workflow means you shouldn't bolt on docs at the end, but continuously update them as you go (docs-driven dev as mentioned earlier).

- **Update the README and Design Docs:** Whenever a major feature is added or a change in architecture is made, reflect that in the README (at least at a high level) or in `docs/DESIGN.md` for more detailed technical decisions. This prevents the "I'll document it later" trap. With vibe steering, you can even enlist Claude's help: *"Summarize the authentication module for the README"* – it can draft a concise description which you can then polish [16] (Claude is aware of your codebase context, so it can describe it in plain English).
- **Inline Documentation:** Encourage writing docstrings and comments in the code, especially for complex logic. You can ask Claude to generate docstrings for functions you wrote: *"Add a Python docstring to this function explaining its purpose and parameters."* This ensures future maintainers (or you, months later) can quickly recall the intent.
- **Knowledge Base & Memory:** Claude has a "Memory" feature to store and retrieve info across sessions [17]. Consider using this if the project is long-running – for example, store the project brief or key design decisions in Claude's memory so it doesn't forget them in later conversations. Also, maintain a `CHANGELOG.md` or use version control tags to record changes over time.

Manual Instructions & Handover: The workflow template can include a section for any manual setup or run instructions that can't be fully automated. For instance, if someone needs to obtain an API key and put it in an `.env` file, note that in a **Setup** section of README. If deployment requires a manual step (like "contact ops team to whitelist an IP"), document it. The aim is a new developer (or your future self) can follow the documented steps and get the project running and understand its design *without having to ask past-you*.

Lastly, consider writing a brief **"User Guide"** if this app is user-facing and complex. This might be in `docs/USER_GUIDE.md` or a `docs/` website. You can generate initial drafts of user documentation by asking Claude to rephrase technical details for a non-technical audience.



An iterative documentation-first approach helps guide development. The figure above illustrates an example cycle of README-driven development: you begin by writing documentation (the README) before coding, then develop the code to meet that spec. As you discover new requirements or features, you update the documentation and design accordingly, then implement the next iteration. This ensures your docs and code evolve together, keeping the project aligned with its intended “vibe” and purpose. By continuously steering with documentation, you avoid losing sight of the big picture or accumulating undocumented “magic” in the code.

Tooling: Claude Integration and Other Utilities

Because you’ll be using VS Code with the Claude extension, it’s important to fully leverage Claude’s features and also integrate any other developer tools that boost productivity.

Claude VS Code Extension Tips:

- **Code Navigation and Context:** Claude Code maintains awareness of your project structure and can answer questions about it [16] . Don’t hesitate to ask it questions like *“Where is the logic for X defined?”* or *“Which files handle the payment processing?”*. Claude can find relevant code or even open the files for you. This is great when jumping back into a project after time away.
- **Using Claude Skills:** We mentioned custom skills earlier. To use them, ensure they are placed in `.claude/skills/<skill-name>/SKILL.md`. Claude will automatically load them when relevant or allow you to invoke via slash command [12] . For example, a skill `scaffold-component` with `name: scaffold-component` in YAML can be invoked by typing `/scaffold-component` in the Claude chat. You might create skills for tasks you do often, such as `/create-api [name]` to generate a boilerplate for a new API endpoint, or `/run-tests` to execute your test suite and report results. Skills can include code (scripts) or just instructions, and you can restrict them or pass arguments as needed [18] [19] .
- **File Operations and Execution:** Claude Code can **edit multiple files**, create new ones, and run shell commands within VS Code [7] . For instance, you can ask Claude to open a new file and fill it with

content, or run `npm install` to add a library. This means you can achieve a lot without leaving your IDE. However, always double-check any destructive operations (like file deletions or running a build) that Claude performs, just in case.

- **Web Browsing and External Info:** If you need information from outside (say, the latest docs of a framework, or an example from Stack Overflow), Claude's web fetch tool can retrieve web content [14] [15]. Use this to your advantage for research. Example: *"Fetch the official Django tutorial for forms"* or *"Search the web for how to implement OAuth with Next.js"*. Claude can pull in those external references (with citations even) so you can make informed decisions with up-to-date info. This helps keep your knowledge current (since you noted that staying updated is important and your internal data might be outdated).

Other Tools and Extensions:

- **Linting & Formatting Extensions:** Ensure you have VSCode extensions for your linter/formatter so that issues are highlighted as you code (or even auto-fixed on save). This real-time feedback complements Claude's assistance.
- **Snippet Managers:** You might still use traditional snippet tools (like VSCode snippets or Emmet for HTML) for things you type often. They can coexist with Claude. For example, Emmet for quick HTML structure, then Claude to fill in specific content.
- **Live Preview / Hot Reload:** Use dev servers (like `vite` or `webpack dev server` for web, etc.) so you can immediately see the effect of Claude's changes in the running app. The faster you see results, the quicker you can refine via prompts.
- **Version Control Integration:** Use Git integration in VSCode. When Claude makes changes, you can use the diff viewer to see exactly what was modified. This is useful both for code review and for learning (it's like seeing a pair-programmer's commit). Commit frequently with clear messages.

By combining Claude's capabilities with these tools, your workflow becomes a powerful hybrid of AI-driven automation and engineer-driven oversight. It's like having a capable assistant always in your IDE, plus the usual power tools of development – make sure to harness both.

Deployment and DevOps

Building the app is half the battle – deploying it reliably is the other. Here your DevOps skills come into play. You can also use AI to assist with setting up the deployment process, while ensuring best practices like infrastructure-as-code and repeatability.

Containerization: Almost all modern apps benefit from being containerized for consistency across environments. Early in the project (once your app can run locally), create a `Dockerfile`. If you're not an expert in the particular stack's Docker setup, ask Claude to draft one: *"Write a Dockerfile for a Node.js Express app using Alpine Linux, exposing port 3000."* Verify the Dockerfile builds and runs. Because you'll possibly orchestrate with Kubernetes, ensure the image runs in a standalone mode (no dev-specific hacks). You might incorporate multi-stage builds (Claude can help optimize the Dockerfile if you prompt, e.g., "reduce the image size").

Kubernetes & Orchestration: If deploying to a K8s cluster, define your Kubernetes manifests or Helm charts. Claude can generate YAMLs for Deployment/Service if given parameters: *"Create a kubernetes deployment for this app with 3 replicas and an ingress on /api."* Double-check things like environment variables, persistent volumes, etc., as AI might not know your exact infra. Since K8s YAML can be verbose, generating with AI saves time, but always test in a safe environment. Use your knowledge to fill gaps (like specific resource requests or network policies required by your org).

CI/CD for Deployment: Extend your CI pipeline to deploy. For example, use a GitHub Actions job to build and push the Docker image on each release tag. AI can assist writing this. Also, consider Claude's integration in CI – there's mention of Claude Code in CI contexts (GitHub Actions integration [20]). In principle, you could have Claude automatically do some checks or even updates in CI, but that's advanced. As a starting point, just ensure your pipeline can deliver artifacts to your deployment environment.

Manual Deployment Steps: Document any manual steps, but strive to automate them. If you find yourself clicking in a cloud console, see if you can script it or at least note it in docs for later automation. Your vibe-steering workflow should treat infrastructure as part of the project code – meaning it's version-controlled and reproducible. Use Infrastructure-as-Code tools (Terraform, CloudFormation, etc.) if applicable. Claude can also help write IaC templates; e.g., *"Generate a Terraform script to provision an EC2 instance with security group for the app"*.

Monitoring & Post-Deploy: After deployment, set up monitoring/logging. This might not be in your initial template, but is part of a complete workflow. Use tools like Prometheus/Grafana or cloud monitors, and ensure the app has health checks. If issues arise in production, you can consult Claude for quick analysis (for instance, feeding log snippets to it and asking what might be wrong). But always treat production with caution – test thoroughly in staging environments to catch problems early.

Iteration, Feedback, and Continuous Improvement

Once a first version of the app is live or feature-complete, the workflow doesn't stop. **Vibe steering** is an ongoing process, especially as you gather user feedback or new ideas.

- **User Feedback Loop:** If it's a user-facing app, pay attention to user feedback on both functionality and design. Where possible, translate this feedback into actionable tasks. *Example:* Users find a feature confusing – you might need to adjust the UI text or workflow. This becomes a new prompt for Claude: *"Simplify the onboarding flow, perhaps by adding a tooltip or tutorial screen for first-time users."* Because you documented the design goals, you can ensure changes remain consistent with the desired vibe.
- **Performance Improvements:** As usage grows, identify any performance bottlenecks (monitoring tools help here). Then use the same develop loop to improve: *"Optimize the database queries in the reports module"* could be a prompt. Claude might suggest adding an index or caching – again, verify and implement.
- **Regular Refactoring:** Periodically, take time to refactor and clean up debt. AI can assist by suggesting more modern approaches or extracting repetitive code into reusable functions. The **common workflows** in Claude docs show examples of using Claude for refactoring safely [21] . For instance, ask *"Refactor this legacy code to use modern hooks API"* (if React) or similar. Use plan mode or step-by-step confirmation for large changes.
- **Stay Updated:** The tech world moves fast, and since you value best practices, keep an eye out for updates (framework updates, new libraries, etc.). You can ask Claude to check if, say, any dependencies have known vulnerabilities or if a new version has come out. (Claude won't

automatically know new versions unless you integrate web search, but you can easily pull release notes from the web with it.)
- **Expand Skills and Automation:** As you refine your workflow, you may identify new repetitive tasks that can be automated. Continue to create or refine Claude skills, scripts, or VSCode tasks. Perhaps you create a skill to generate a **release notes** draft by scanning commit messages (Claude can do this by reading git logs and formatting a summary). In fact, Claude Code can help "write release notes in a single command" as advertised [4], which is a great example of vibe steering for project management tasks.

Retrospective Questions: After each project (or periodically), ask yourself: - *What went well in the workflow? What saved the most time?* (Do more of this) - *What issues did I encounter?* (e.g. AI gave a bad code that took long to debug, or design was still lacking somewhere) - *How can I adjust the template or tools to avoid those issues in the next project?* (Maybe add a new checklist item, or a new skill, or choose a different library).

In essence, treat the workflow itself as an evolving product – continuously improve your vibe steering process.

Conclusion

By following this vibe-steering workflow, you create a repeatable template for software projects that balances creativity and efficiency. You leverage Claude's AI muscle to **accelerate development** (planning, coding, documenting, testing) but **anchor the process in best practices** (solid architecture, component libraries, thorough documentation and testing). The result should be well-written applications that *feel* professionally designed and coded, even if you leaned on AI for the heavy lifting in parts.

Remember that the "vibe" of an application comes from many layers – the code quality, the performance, the user interface, and the overall user experience. By systematically addressing each layer with the help of AI and good engineering practices, you ensure that the final product isn't just built faster, but is something you're proud of and others find delightful to use.

Lastly, don't be afraid to **ask questions** (to yourself and to Claude) at every step. The prompt-driven development means that articulating what you need is half the battle – the clearer your questions and instructions, the better the outcomes [3]. Happy vibe coding, or rather, vibe *steering*!

Sources: The workflow and tips above were compiled referencing best practices and tool capabilities from Claude's documentation and developer community insights. Key references include Claude's official docs on coding assistance [3] [22] and skills extension [12] [18], an industry perspective on "vibe coding" pitfalls [1] [2], advice from developers on using component libraries for better design [5], and the README-driven development approach [6] which emphasizes starting projects with solid documentation. These sources reinforce the elements of the workflow, from AI integration to design and documentation strategies.

---

[1] [2] Vibe coding: What is it good for? Absolutely nothing • The Register
https://www.theregister.com/2025/11/24/opinion_column_vibe_coding/

[3] [4] [7] [10] [11] [16] [20] [22] Claude Code overview - Claude Code Docs
https://code.claude.com/docs/en/overview

[5] I don't know who needs to hear this, but just use a component library. : r/nextjs
https://www.reddit.com/r/nextjs/comments/160sfpp/i_dont_know_who_needs_to_hear_this_but_just_use_a/

[6] README driven development - DEV Community
https://dev.to/gregmartinez44/readme-driven-development-373k

[8] [21] Common workflows - Claude Code Docs
https://code.claude.com/docs/en/common-workflows

[9] [14] [15] [17] Features overview - Claude API Docs
https://platform.claude.com/docs/en/build-with-claude/overview

[12] [13] [18] [19] Extend Claude with skills - Claude Code Docs
https://code.claude.com/docs/en/skills